Performance Modeling for Dense Linear Algebra

Elmar Peise and Paolo Bientinesi

AICES, RWTH Aachen

Schinkelstr. 2

52062 Aachen, Germany

{peise,pauldj}@aices.rwth-aachen.de

Abstract—It is well known that the behavior of dense linear algebra algorithms is greatly influenced by factors like target architecture, underlying libraries and even problem size; because of this, the accurate prediction of their performance is a real challenge. In this article, we are not interested in creating accurate models for a given algorithm, but in correctly ranking a set of equivalent algorithms according to their performance. Aware of the hierarchical structure of dense linear algebra routines, we approach the problem by developing a framework for the automatic generation of statistical performance models for BLAS and LAPACK libraries. This allows us to obtain predictions through evaluating and combining such models. We demonstrate that our approach is successful in both single- and multi-core environments, not only in the ranking of algorithms but also in tuning their parameters.

I. INTRODUCTION

For a large class of dense linear algebra operations, such as the solution of least squares problems and linear systems, not one but many algorithms exist. While mathematically they are equivalent, their performance depends —in different ways— on factors such as the target architecture, the underlying libraries, and the problem size. The prediction of the best performing algorithm in a given scenario is a challenging task. Our goal is to *rank the algorithms* according to their performance and to determine their optimal configuration *without executing them*.

As a motivating example, we consider the inversion of a lower triangular matrix ($L \leftarrow L^{-1}$), for which there exist four blocked algorithms, all equivalent in exact arithmetic, but with different performance signatures. Each such algorithmic variant depends on one parameter, the block-size b, which determines the stride in which the matrix is traversed.

In Figure I.1 we plot their efficiency —the relative measure of the machines resource utilization— when executed on one core of an Intel Harpertown E5450; [1] the block-size is fixed to 96 and the matrix size n varies. The results show noticeable differences in performance between algorithms: Variant 4 (●) is significantly slower than the others, while, especially for large matrices, variant 3 (●) is the most efficient. In Figure I.2, we fix $n = 1000$ and let b vary; in all variants, the efficiency decreases for small and large block-sizes. For variants 1 (●), 2 (●), and 3 (●) the optimal choice of b is close to 100.

This example shows that in order to reach high efficiency, it is crucial to both single out the right algorithmic variant, and optimize the block-size. Due to the complexity of the

---

[1] The algorithms were implemented in C, compiled with ICC version 12.0, and linked to Intel's MKL version 10.2.6.
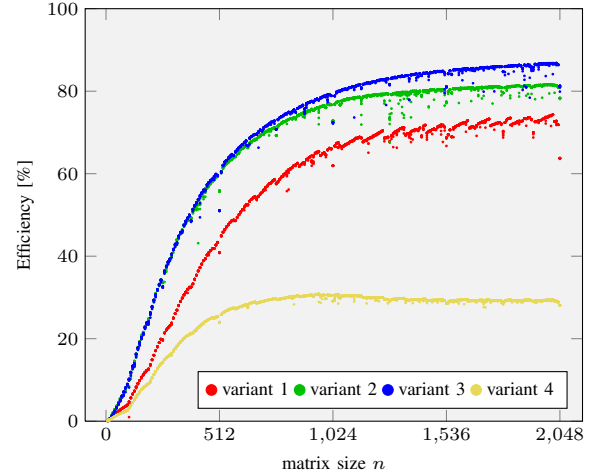

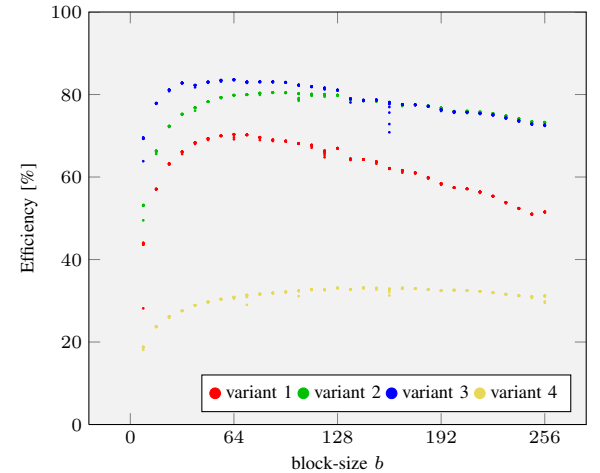Fig. I.1: Inversion of a lower triangular matrix: Efficiency as a function of the problem size.

Fig. I.2: Inversion of a lower triangular matrix: Efficiency as a function of the block-size.

architecture and the memory access patterns, it is virtually impossible to perform these tasks only by analyzing the mathematics of the algorithms. Indeed, experience tells us that the best choice heavily depends on the computational kernels used, such as BLAS, on the processor architecture, and the matrix size; changing any of these factors may lead to entirely different performance behavior.

In this article, we detail a strategy based on the analysis of the BLAS routines upon which the target algorithms are built; we introduce a tool that, using measurements, creates

performance models for BLAS kernels and stores them permanently in a repository. When faced with a set of algorithms, the models are evaluated and combined to predict the algorithms' performance. These predictions allow us not only to accurately rank the algorithmic variants, but also to determine the optimal algorithmic block-size.

Several different approaches to performance modeling in dense linear algebra exist; some notable examples are given in the following. Cuenca et al. developed a system of self-optimizing linear algebra routines (SOLAR) [1]; every routine is associated with performance information, which is hierarchically propagated to higher level routines in order to tune them. Dongarra et al. proposed an approach for parallel software such as HPL and ScaLAPACK [2]; they employ sampling and polynomial fitting to construct models in order to extrapolate the performance of routines for larger problems and higher parallelism. Iakymchuk et al. model the performance of BLAS analytically based on memory access patterns [3]; while their models represent the program execution very accurately, constructing them requires a high level of expertise of both routines and architecture.

In contrast to the aforementioned approaches, we aim at the automatic generation of accurate models for BLAS routines, which constitute the building blocks of a multitude of algorithms in linear algebra. Our main goal is not to obtain accurate prediction for these algorithms, but rather to correctly rank them and tune their configuration.

This article is structured as follows. In Section II, we discuss the performance of dense linear algebra routines. In Section III, we introduce the Modeler, a tool that automatically generates analytical performance models. Predictions and ranking are discussed in Section IV, and in Section V we draw conclusions.

II. PERFORMANCE

In this section, we discuss the concept of performance in the context of dense linear algebra, and introduce *the Sampler*, a performance measurement tool for linear algebra routines.

A. Performance Metrics

In the following, the term performance is used broadly to cover a set of *performance metrics* that describe certain aspects of a routine execution, such as timings, instruction counts, and cache accesses. The metrics are either directly obtained from hardware performance counters or are quantities computed from them. The most fundamental performance counter — the time stamp counter— is provided by a register that is incremented once per CPU cycle. It is accessed through the x86 instruction `RDTSC` and serves as a cycle-accurate timer; we refer to this metric as *ticks*. In order to access more CPU performance counters, we use the Performance Application Programming Interface (PAPI) [4][2]. PAPI provides functions to configure, initialize, and read up to 107 counters, but usually only a subset of which are available in a given system.
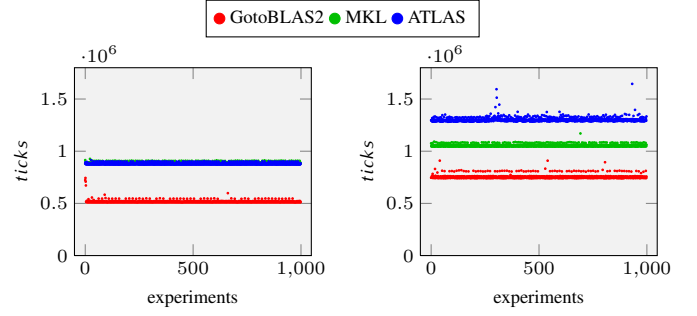
---

[2]PAPI version 4.2.1.0.

Fig. II.1: Repeated execution of `dtrsm`: In-cache (left) and out-of-cache (right) operands.

In this article we focus on the highly accurate time metric *ticks*. In additions, we use the derived metric *efficiency*, representing the relative resource utilization:

$$efficiency = \frac{flops}{ticks \cdot fips}.$$

This measures how efficiently an operation that performs *flops* floating point instructions[3] uses the CPU's ALUs, which can perform up to *fips* floating point instructions per cycle.

B. Performance of dense linear algebra routines

At this point, we are interested in the performance of dense linear algebra routines, such as BLAS or unblocked algorithms, that act as building blocks for higher level algorithms. Our first objective is to build performance models for such building blocks.

For a given architecture, we regard the performance of a routine as a function of the arguments. Apart from the buffers for matrices and vectors, all the arguments are simple to represent in such a function, since they are basic data types such as characters, integers, and floating point numbers. Since the instructions performed by the dense linear algebra routines we consider are mostly independent of the input data, we can reduce the information needed for these arguments to their size and storage location in the memory hierarchy.

Regarding memory locality, we distinguish two cases: in-cache and out-of-cache. *In-cache* refers to the situation where all matrices are as close to the CPU as possible, that is, in the lowest cache level that can accommodate them. Since the access time is minimized, this scenario leads to the best performance the routine can attain. *Out-of-cache* refers to the opposite situation, where the matrices reside in main memory, thus causing costly data transfers. Since the loading and storing of data might result in memory stalls, the overall performance is often inferior than when data resides in cache.

To study the influence of memory locality and the reproducibility of measurements, let us consider a repeated execution of the BLAS routine `dtrsm` ($B \leftarrow A^{-1}B$, A triangular). The interface is

```
        side  uplo  transA  diag   m     n
dtrsm( R ,  L ,   N  ,  U , 512, 128,
```

---

[3]A fused multiply add operation $a \leftarrow b + c \cdot d$ is counted as a single floating point operation, since it is one instruction and processed as such by the CPU.

```
       alpha   A   ldA   B   ldB
      0.37,   A,  256,  B,  512),
```

corresponding to the operation $B \leftarrow 0.37BA^{-1}$, where $A \in \mathbb{R}^{128 \times 128}$ is lower triangular with leading dimension $\texttt{ldA} = 256$, and $B \in \mathbb{R}^{512 \times 128}$ with $\texttt{ldB} = 512$. In our experiment, this operation is repeatedly executed on one core of our Harpertown, using the high-performance BLAS implementations GotoBLAS2, MKL, and ATLAS. Notice that the first invocation of a BLAS routine is always notably (in our case more than one order of magnitude) slower than the following one, due to the initialization of BLAS, which happens at the first invocation of the library. Neglecting these first measurement outliers, the performance measurements of the routine executions with both in-cache and out-of-cache arguments are shown in Figure II.1. As expected, in-cache corresponds to higher performance across all implementations, while the increase in execution time for out-of-cache varies from one implementation to the other. In our study —in which the performance of algorithms is obtained through models of the algorithms' components— memory locality will play a big role.

In addition to the influence of memory locality, we observe fluctuations in the performance measurements of about 8%. For this reason, we do not consider the routine's performance to be one number but a probabilistic distribution. To express the performance in numbers, we select certain properties of this distribution, such as minimum, average, standard deviation, and median.

C. The Sampler

To facilitate the acquisition of performance measurements, we wrote the Sampler, a flexible lightweight performance measurement tool. Written in C, the Sampler directly interfaces with libraries such as BLAS or LAPACK. Its configuration allows to choose between different memory locality situations. Given routine names and arguments in the form of tuples, such as $(dtrsm, R, L, N, U, 512, 128, 0.37, A, 256, B, 512)$ (A and B specify the sizes of the operands), the Sampler measures and reports the performance of the routines; this entails collecting multiple samples and extracting statistical information.
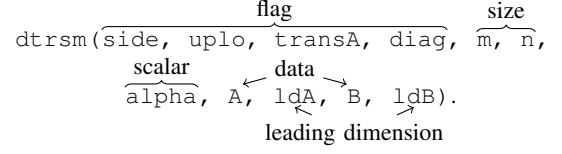
III. MODELING

With the measurements obtained by the Sampler, we want now to construct analytical performance models. Here we introduce the *Modeler*, a tool which interacts with the Sampler and automatically generates performance models. These models form the base for the performance prediction and algorithm ranking (Section IV).

A. Preliminary Experiments

Most BLAS routines accept 10 or more arguments; LAPACK's routines have easily twice as many. In building performance models, if we blindly treated all the arguments equally, we would originate 10+ dimensional models, which would result in either impractical execution times or sloppy accuracy. To avoid this curse of dimensionality, we analyze how different arguments types affect performance, and in our models we only account for a subset of the arguments.

Here we focus on the dependence of performance on the BLAS arguments. Again, we use $\texttt{dtrsm}$ as an example; the arguments of BLAS routines can be classified as follows:

$$\texttt{dtrsm}(\overbrace{\texttt{side, uplo, transA, diag,}}^{\text{flag}} \overbrace{\texttt{m, n,}}^{\text{size}}$$
$$\overbrace{\texttt{alpha,}}^{\text{scalar}} \overbrace{\texttt{A, ldA, B, ldB}}^{\text{data}}).$$
$$\underbrace{}_{\text{leading dimension}}$$

Flag arguments take one of only two values (e.g., $\texttt{side} \in \{\texttt{L}, \texttt{R}\}$); *size* arguments contain the dimensions of the matrix and vector operands; *scalar*s are floating point numbers, which scale the operands; *data* are (pointers to) the buffers in which the operands are stored; *leading dimension*s define the distance in memory between two horizontally adjacent matrix entries.

Due to the following reasons, for our purposes we can disregard all but flag and size arguments.

- Scalar arguments are usually set to 1 or -1. Neither of these values requires any floating point operations to perform the scaling, thus not affecting performance. Even other values for scalar arguments have an insignificant influence on performance, since they only affect a lower order term of the operation count.
- As discussed in Section II-B, only the size and storage location of vector and matrix arguments are relevant for performance. The sizes of these operands are covered by the size and leading dimension arguments. As for the storage locations, we will construct separate models for different memory locality scenarios, so that we can entirely ignore data arguments within one model.
- In practice, the leading dimensions are either equal to the size of a corresponding input matrix or larger. While the difference between these two scenarios can influence performance, we only need to consider the latter: within our targeted algorithms, the BLAS routines are invoked on parts of a large input matrix of constant dimension. Hence, throughout the model generation, all leading dimension arguments are set to 2500.

Next, we study the influence of the flags and the size arguments on the performance of BLAS routines.

1) Flag Arguments: All types of flag arguments encountered in BLAS appear in the signature of $\texttt{dtrsm}$: $\texttt{side} \in \{\texttt{L}, \texttt{R}\}$ defines from which side B is multiplied by A^{-1}; $\texttt{uplo} \in \{\texttt{L}, \texttt{U}\}$ states if A is lower or upper triangular; $\texttt{transA} \in \{\texttt{N}, \texttt{T}\}$ indicates whether A or its transpose A^T is to be used; when set to $\texttt{U}$, $\texttt{diag} \in \{\texttt{N}, \texttt{U}\}$ declares that A is unit triangular.

In Figure III.1, we report on a series of experiments in which we look at the performance for all possible combinations of the flags;[4] the remaining arguments are fixed as follows:

```
        side    uplo   transA   diag    m     n
  dtrsm(side, uplo, transA, diag, 256, 256,
        alpha  A   ldA   B    ldB
       0.5 ,  A,  256,  B,  256).
```

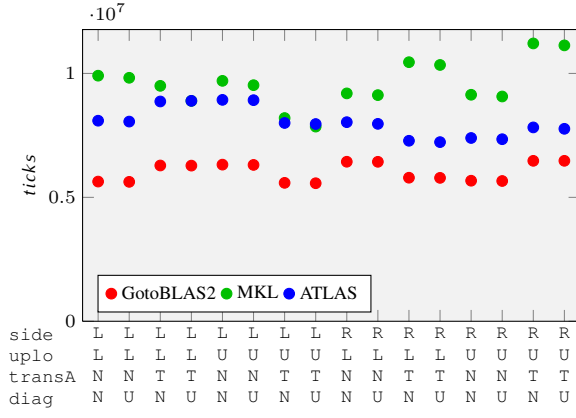[4]Using one core of an AMD Opteron Processor 8356 running at 2.30GHz.

Fig. III.1: dtrsm: *ticks* as a function of the discrete arguments.

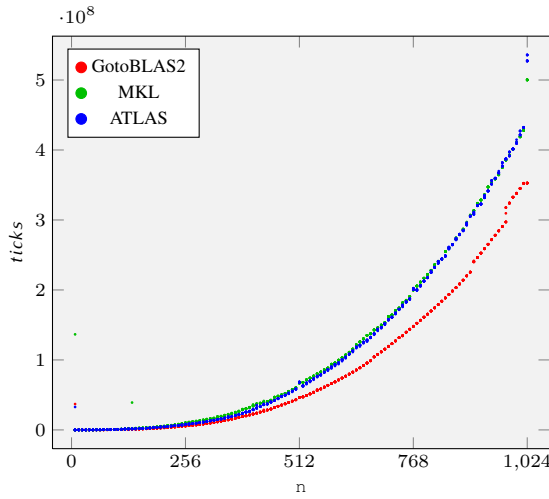Fig. III.2: dgemm: *ticks* as a function of the size arguments.

Fig. III.3: dgemm: Distance between least-squares fitting and original data (Figure III.2).

The only common feature across all implementations is that diag only has a minor impact on performance. No clear pattern arises to relate the performance of two or more arguments. This may be due to different argument values leading to the execution of distinct code branches. We conclude that in our models, with the exception of diag, we should treat all combinations of argument values separately.

2) Size Arguments: We consider the invocation

```
        side   uplo  transA  diag   m   n
dtrsm( L ,   L ,    N  ,   N , n, n,
       alpha   A   ldA   B   ldB
        0.5 ,  A,   n ,  B,  n ),
```

where n varies between 8 and 1024. To avoid the influence of small scale fluctuation, we only consider values of n that are multiples of 8. Measurements for different BLAS libraries are shown in Figure III.2.

At first sight, the measurements follow a quadratic behavior, in line with the routine's complexity. For each of the three BLAS implementations, we construct a quadratic polynomial p that best approximates the measurements through least squares fitting. In Figure III.3 we plot the difference between p and the original measurements; it becomes apparent that in none of the three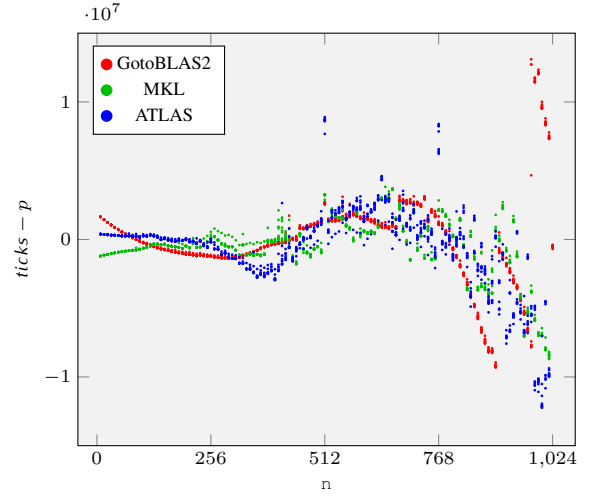 cases, a quadratic approximation represents the performance accurately. However, the plot shows some degree of structure; this is especially visible for GotoBLAS2 (●), where there are intervals with polynomial behavior separated by jumps or kinks. We can make out the following intervals: 0 – 300, 300 – 650, 650 – 900, 900 – 1000, and 1000 onwards. A similar behavior can be found in the other implementations, although the higher fluctuations in their measurements make the jumps less visible.

This experiment teaches us that a single polynomial cannot accurately represent the performance of a routine, even when all the arguments are fixed, and only the size varies. As a consequence, the Modeler will generate models in the form of piecewise multivariate polynomials.

B. The Targeted Models

A performance model represents the performance of a routine for a fixed implementation, system, and memory locality situation. Given a set of valid routine arguments, the model provides estimates on the expected performance in the form of statistical quantities, such as minimum, average, standard deviation, and median.

Internally, our models operate as follows. In order to avoid the curse of dimensionality, only a subset of the routine's arguments are selected; these are the model *parameters*. We distinguish between two types of parameters: flags, corresponding to flag arguments, and integer parameters, corresponding to size (and possibly leading dimension) arguments.

In our models, each combination of flags is treated separately. In a model with 3 flag parameters, with 2 possible values each, this would lead to $2^3 = 8$ separate *submodels*, representing the performance dependence on the integer parameters.[5]

Each submodel is essentially a vector-valued multivariate piecewise polynomial in the following sense. The integer parameters span a multidimensional space, which is covered

---

[5]Since there are no more than 4 flag arguments in BLAS routines, the number of submodels stays well within manageable bounds.

by rectangular regions, in which the behavior is represented by polynomials. Each polynomial is vector valued with one value for each statistical quantity.

When the model is used to estimate the performance for given routine arguments, the following happens: (1) the model parameters are extracted; (2) the submodel corresponding to the combination of flags is identified; (3) the region containing the integer parameter point is found; (4) the polynomial corresponding to the region is evaluated, yielding the estimates.

C. The Modeler

In the previous section, we have described the structure of our targeted performance models. We now introduce the Modeler, a tool that generates these models automatically.

We skip the technicalities that arise from creating a separate model for each combination of flags and instead focus on the generation of piecewise polynomials to model the dependence of performance on integer parameters. The objective of the Modeler is to attain accuracy automatically and with as few measurements as possible. Moreover, although within BLAS at most three integer parameters are encountered, the Modeler is designed for arbitrary dimension.

Polynomial Fitting through Least Squares: The approximation of a set of sampling results by polynomials through least squares fitting is a fundamental task of the modeling process. A set of n coordinate value pairs $(\mathbf{x}_i, v_i)$ are fitted with a polynomial p of limited order, such that

$$\sum_{i=1}^{n} (p(\mathbf{x}_i) - v_i)^2$$

is minimized. To solve this least squares problem, we use the function `linalg.lstsq()` provided by Python's SciPy package, which is based on singular value decomposition.

The accuracy of a polynomial approximation p is determined by the local errors $e_i = p(\mathbf{x}_i) - v_i$. While the used least squares method minimizes $\sum_{i=1}^{n} e_i^2$, we use the *maximum relative error* across all $\mathbf{x}_i$:

$$e_{\text{relmax}} = \max_{1 \leq i \leq n} \frac{|e_i|}{v_i}.$$

To create the vector valued polynomial for the statistical quantities, each quantity is separately fitted with a polynomial. The error e_{relmax} of one quantity is picked to represent the accuracy of the whole polynomial. In the following, we use the median for this purpose.

1) Model Expansion: We now introduce the first of two modeling strategies, *Model Expansion*. The piecewise polynomial is created according to the following steps.

- The first objective is to build a model for a small region in a corner of the parameter space; this is accomplished by fitting a small set of measurements.
- This initial region is then expanded as much as possible, by taking new measurements, integrating them in the model, and checking that
  a) the polynomial's approximation error is below a given threshold, and

Fig. III.4: Sequence of steps in the construction of piecewise models through Model Expansion.

  b) the region stays within the boundaries of the parameter space.
- Once a region cannot be extended further —because of either a) or b)— new adjacent regions are generated and expanded.

The process is repeated until the whole parameter space is covered.

An example of Model Expansion in two dimensions is given in Figure III.4. The rectangular domain is filled starting with region 1 (■) in the bottom-left corner of the domain. This region is expanded in both directions until its accuracy reaches the threshold (■). Two new adjacent regions 2 and 3 (■) are then created —together with the associated samples— and their expansion begins. Assuming that region 2 was expanded as far as possible (■), region 4 (■) is then generated. Once the expansion for region 3 (■) is complete, the neighboring regions 5 and 6 (■) are created. These 6 models cover the full parameter space, therefore the process terminates.

2) Adaptive Refinement: The second strategy to generate piecewise models is based on adaptive refinement. The idea is to begin with a simple and regular model constructed from a coarse grid of samples across the whole parameter space; the quality of such a model is then evaluated. If insufficient, the region is split and the model is refined by locally increasing the sample grid resolution. These steps are applied recursively to the refined regions until either the accuracy reaches a satisfactory level across the whole domain, or a given resolution limit is reached.

An example of Adaptive Refinement for a two dimensional domain is shown in Figure III.5. The polynomial approximation for the initial region spanning the entire parameter space is very inaccurate (■, 1st square on the left). Therefore it is refined, generating four new regions, and new measurements are obtained to create four polynomials (2nd square on the left). Now, the error in the top right quadrant (■) is already below the threshold (■); the other quadrants are not accurate enough and are further refined (3rd square). In the next iteration, several regions are below the desired error threshold; the
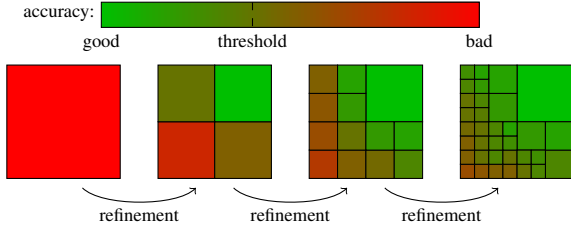
Fig. III.5: Sequence of steps in the construction of piecewise models through Adaptive Refinement.

others are refined once more (4th square). Although some of the resulting polynomials are still above the desired level of accuracy, they are accepted anyway, because their size does not allow further refinement.

D. Results

Having introduced two modeling strategies, here we compare the resulting models, both in terms of speed and accuracy. We consider again the solution of a triangular system $B \leftarrow A^{-1}B$ as testbed:

```
dtrsm(side, uplo, transA, diag, side, m, n,
        alpha, A, ldA, B, ldB).
```

The interface of this routine contains four flags (side through diag), two size arguments (m and n), one scalar argument (alpha), and operates on two matrices (A and B with corresponding leading dimensions ldA and ldB). Out of these arguments, our models account for

- the flag parameters side, uplo, and transA, and
- the integer parameters m and n.

The integer parameters vary in the range $[8 - -1024]$ and define the parameter space; the flags are $(\text{side}, \text{uplo}, \text{transA}) = (\text{L}, \text{L}, \text{N})$; the values of the remaining arguments are: diag = N, alpha = 0.5, and ldA = ldB = 2500. We use the in-cache configuration of the Sampler and GotoBLAS2[6]

1) Model Expansion: This approach accepts several configuration options:

- the relative error bound ε;
- the direction of expansion $d \in \{\nearrow, \swarrow\}$;
- the initial size of regions s_{ini}.

We illustrate the influence of such options on the generation by presenting models obtained with different settings. The plots in Figure III.6 show how differently these models cover the parameter space and display the relative errors.

In Figure III.6a, we used the configuration:

- the error bound is $\varepsilon = 10\%$;
- the direction of expansion is $d = \nearrow$;
- new regions are initially of size $s_{ini} = 128$.

Smaller and less accurately modeled regions are generated towards the left side of the parameter space. Towards the top right corner, the regions become larger and the relative error decreases. In this part of the parameter space, we also find
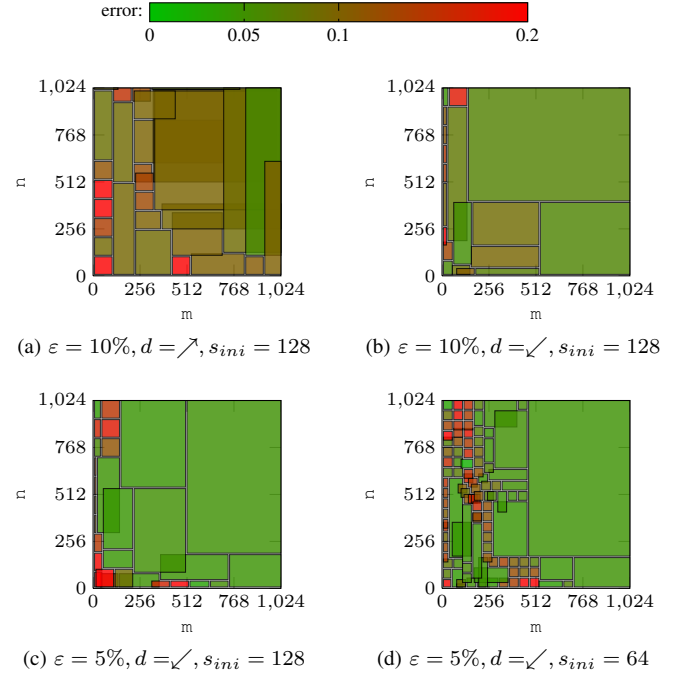
(a) $\varepsilon = 10\%, d = \nearrow, s_{ini} = 128$

(b) $\varepsilon = 10\%, d = \swarrow, s_{ini} = 128$

(c) $\varepsilon = 5\%, d = \swarrow, s_{ini} = 128$

(d) $\varepsilon = 5\%, d = \swarrow, s_{ini} = 64$

Fig. III.6: Model Expansion for dtrsm.

several areas which are modeled by two or more overlapping regions[7].

In Figure III.6b instead, we let the Modeler expand along the direction $d = \swarrow$. We observe the following changes:

- especially towards the top right corner, the generated regions are larger;
- these regions are of higher accuracy compared to the previous model, although the error bound was not modified;
- fewer regions overlap.

The average relative error improves from 10.4% to 6.99%, while the number of required sampling points decreases from $65,220$ to $32,680$. We noticed that in general, it is preferable to expand the models towards the origin ($\swarrow$).

In Figure III.6c, we reduced the error bound to $\varepsilon = 5\%$. As a result, the average model error improves from 6.98% to 5.79%. This comes at the cost of an increase in the number of samples: from $32,580$ to $53,550$. As in the previous cases, the accuracy of the models decreases as the parameter values become smaller; the least accurate models appear for small values of m.

Finally, in Figure III.6d we decreased the size of the initial models from $s = 128$ to $s = 64$. Interestingly, even though the model now makes use of $138,290$ samples and has a finer resolution, the average error increases from 5.79% to 5.96%. This is due to the presence of fine structures, which are revealed by the smaller regions; in the previous models, the sampling was coarser such artifacts were not included in the computation of the approximation error. From this we conclude that an index of accuracy generated from used samples is not always sufficient to capture the global quality of a model.

[7]When the model is evaluated at a point covered by multiple regions, the most accurate model is selected.

(a) $\varepsilon = 10\%, s_{min} = 64$

(b) $\varepsilon = 5\%, s_{min} = 64$

(c) $\varepsilon = 10\%, s_{min} = 32$

(d) $\varepsilon = 5\%, s_{min} = 32$
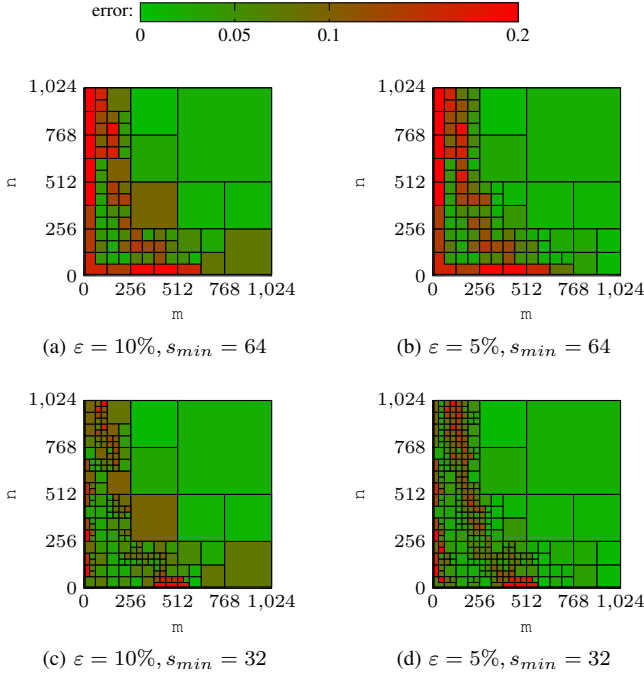
Fig. III.7: Adaptive Refinement for `dtrsm`.

Fig. III.8: Model Expansion vs. Adaptive Refinement.

2) Adaptive Refinement: The generation of models is governed by two options:

- the relative error bound ε, and
- the minimum region size s_{min}.

The regions resulting from different values of these options are shown in Figure III.7.

The first model in Figure III.7a was generated with an error bound of $\varepsilon = 10\%$ and a minimum region size of $s = 64$. The result shows an overall distribution of regions similar to the model in Figure III.6d of Model Expansion: Smaller and less accurately modeled regions are predominant for smaller parameter values — especially for m. Regions on the finest level are not generated on the lower edges of the parameter space (beginning at 8), since they would be smaller than s_{min}. The seemingly rectangular regions are parts of larger regions that were only partially refined.

In Figure III.7b, the error bound was decreased to $\varepsilon = 5\%$. For such an accuracy to be attained, several of the regions from the previous model are further refined, especially on the left side. The increased number of regions is covered by $81,890$ samples — $20,010$ more than previously. The higher accuracy requirement leads to a decrease in the average error from 7.21% to 6.32%.

In the next two experiments, Figures III.7c and III.7d, we decreased the minimum region size to $s = 32$, maintaining the error bound ε to 10% and 5%, respectively. This leads to the generation of many tiny regions. The error bound of 10% (5%) leads to an average error of 4.29% (3.17%) at the cost of $134,160$ ($227,820$) samples.

3) Comparison: Figure III.8 displays, for both modeling strategies, how many samples are needed to generate models of a certain accuracy. The models presented in the previous sections are labeled according to Figures III.6 and III.7. We
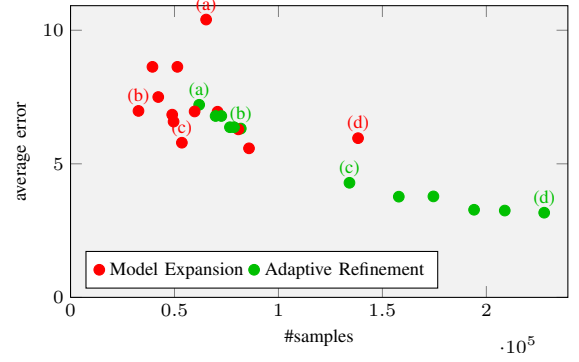
are interested in models that attain a high degree of accuracy with a small number of samples; these are the points laying in the bottom left border of the convex hull (envelope) of all the points.

For relatively few samples, Model Expansion generates more accurate models ((b) and (d)). However, one should keep in mind that these models do not represent the fine scale behavior of *ticks* very well. If one is willing to use a larger number of samples, Adaptive Refinement generates more accurate models ((c)). If the number of samples is not an issue, this method has the potential to generate very accurate models ((d)).

In the experiments performed in the rest of the paper, we used Adaptive Refinement with configuration (c): $\varepsilon = 10\%$ error bound and $s_{min} = 32$ minimum regions size. This configuration is a good compromise between the model accuracy and the number of samples.

IV. PREDICTION, RANKING AND OPTIMIZATION

We are finally ready to tackle our main goal: ranking linear algebra algorithms by performance models. In order to predict the performance of an algorithm, we start by analyzing its sequence of subroutine invocations. We use the models automatically generated by the Modeler to estimate the performance of such invocations. These estimates are then accumulated, resulting in the prediction of the algorithm's performance. The probabilistic nature of the performance model allows us to give detailed information on the expected ranges of the algorithm's performance.

A. Triangular Inverse $L \leftarrow L^{-1}$

We consider four blocked algorithms for the inversion of a triangular matrix. All these algorithms partition L into 6 submatrices as

$$L = \begin{pmatrix} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ L_{20} & L_{21} & L_{22} \end{pmatrix}.$$

The central matrix L_{11} is of size $b \times b$ (the block-size); the size of the matrix L_{00} is initially 0×0, and as the algorithm unfolds, it increases in steps of size b, until L_{00} spans the whole matrix L; the size of L_{22} decreases accordingly; similarly, the sizes of the offdiagonal matrices are entirely determined by those
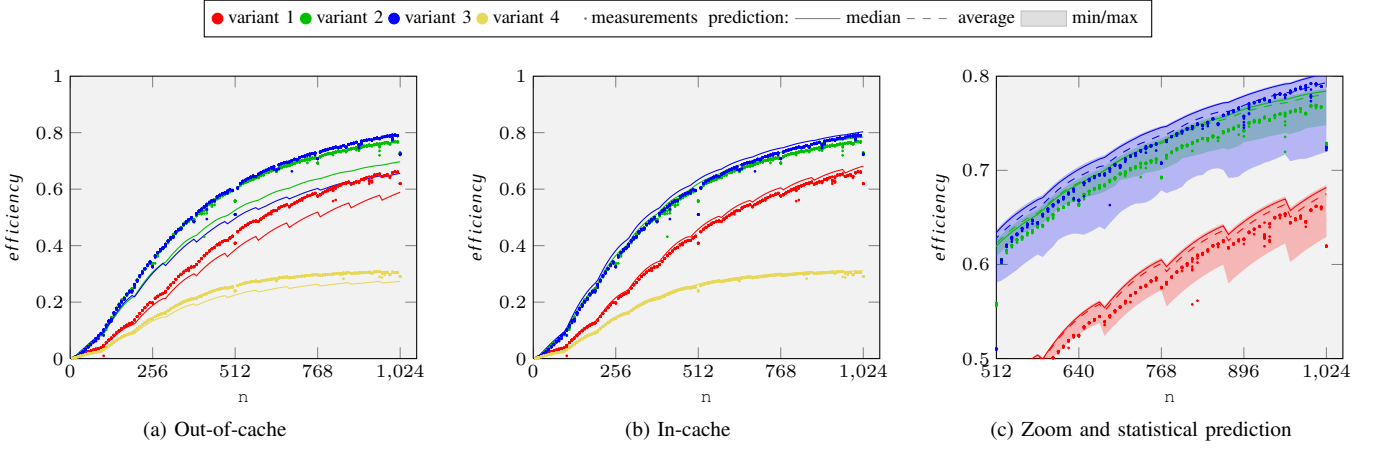
Fig. IV.1: `trinv`: Performance predictions vs. observations.

of L_{00}. At each step of this matrix traversal, a sequence of update statements is performed on the submatrices, such that L_{00} contains a fully computed portion of L^{-1}. Once L_{00} spans all of L, L^{-1} has been computed in place.

The four algorithmic variants presented here differ in their update statements:

Variant 1
$L_{10} \leftarrow L_{10}L_{00}$
$L_{10} \leftarrow -L_{11}^{-1}L_{10}$
$L_{11} \leftarrow L_{11}^{-1}$

Variant 2
$L_{21} \leftarrow L_{22}^{-1}L_{21}$
$L_{21} \leftarrow -L_{21}L_{11}^{-1}$
$L_{11} \leftarrow L_{11}^{-1}$

Variant 3
$L_{21} \leftarrow -L_{21}L_{11}^{-1}$
$L_{20} \leftarrow L_{21}L_{10} + L_{20}$
$L_{10} \leftarrow L_{11}^{-1}L_{10}$
$L_{11} \leftarrow L_{11}^{-1}$

Variant 4
$L_{21} \leftarrow -L_{22}^{-1}L_{21}$
$L_{20} \leftarrow -L_{21}L_{10} + L_{20}$
$L_{10} \leftarrow L_{10}L_{00}$
$L_{11} \leftarrow L_{11}^{-1}$

They are built on top of the BLAS routines `dgemm`, `dtrsm`, and `dtrmm`; the last statement in each algorithm is a recursive call to an unblocked version of the same algorithm. They have the following signatures: `trinvi(n, L, ldL, blocksize)`. We consider their performance with the arguments

$$\text{trinv}i(\underset{n}{n}, \underset{L}{L}, \underset{n}{n}, \underset{96}{\text{blocksize}}),$$

varying the matrix size n $\in \{8, 16, \dots, 1024\}$. We consider the performance metric *efficiency*, which is computed from *ticks* as follows:

$$efficiency = \frac{\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n}{2 \cdot ticks}.$$

For our performance prediction, we use performance models for `dtrsm`, `dtrmm`, `dgemm`, and the unblocked versions of the blocked algorithms[8]. The models are generated by the Modeler, with the configuration selected in Section III-D3. For each algorithm execution, we consider the list of subroutine invocations, consisting of calls to these routines. For instance, the execution of variant 1 on a matrix of size 250 with block-size 100 produces the following invocations:

[8]Since the unblocked versions are only invoked on small matrices, their models are limited to values of n below 256.

```
dtrmm(R, L, N, N, 100, 0, 1, L00, 250, L10, 250)
dtrsm(L, L, N, N, 100, 0, -1, L11, 250, L10, 250)
trinv1(100, L11, 250, 1)
dtrmm(R, L, N, N, 100, 100, 1, L00, 250, L10, 250)
dtrsm(L, L, N, N, 100, 100, -1, L11, 250, L10, 250)
trinv1(100, L11, 250, 1)
dtrmm(R, L, N, N, 50, 200, 1, L00, 250, L10, 250)
dtrsm(L, L, N, N, 50, 200, -1, L11, 250, L10, 250)
trinv1(50, L11, 250, 1).
```

Each invocation corresponds to the evaluation of the corresponding performance model; the results are then accumulated, thus generating a performance prediction.

1) Matrix size: Figure IV.1 contains the predictions for the four algorithm, with varying matrix size. The left and middle panels refer to in-cache (IV.1b) and out-of-cache (IV.1a) scenarios, respectively. Since the memory locality of an actual execution is somewhere in between these two scenarios, neither of the predictions matches the measurements perfectly: in-cache overestimates the efficiency of the algorithms, while out-of cache underestimates it. At this moment we do not yet attempt the construction of models matching the exact memory locality scenario of each algorithm; therefore in the following we use the upper bound on efficiency resulting from the in-cache models. This prediction ranks exactly all variants for all problem sizes.

The previous discussion referred to predictions for the median of the performance. In Figure IV.1c we instead look at average, minimum, and maximum efficiency; in order to visualize the interesting features, we only present the top right portion of the graph ($n \geq 512$ and $0.5 \leq efficiency \leq 0.8$). The ranges between minimum and maximum ( ) cover almost all the observations of the corresponding algorithms, giving a good idea of the expected results; their height is due to the presence of outliers.

The average (- - -) is closer to the measured algorithm performance than the previously used median. Nevertheless, relying on the average predictions, is dangerous, since they are obtained for models generated with an error bound on the median and are influenced by outliers.

Altogether, we predicted performance for varying matrix sizes with highly satisfactory results.
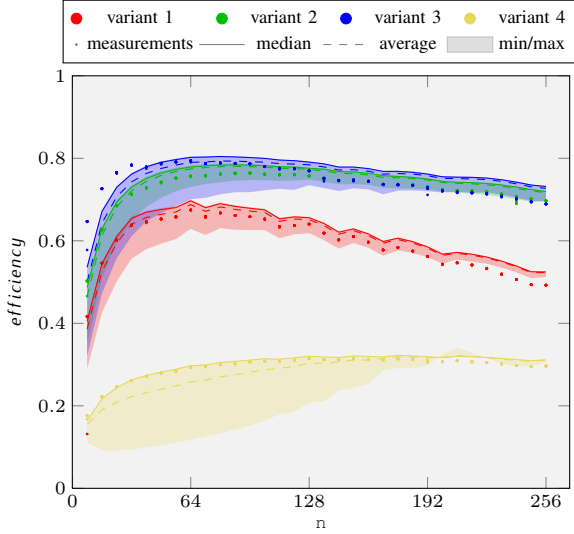
Fig. IV.2: Block-size optimization for `trinv`.

Fig. IV.3: `trinv`: Predictions and observations on 8 cores.

2) Block-size: We now turn our attention to our second point of interest: tuning the block-size to yield the best efficiency. For this purpose, we fix the matrix size to $n = 1000$ and vary the block-size:

```
                 n    L    ldL     blocksize
        trinvi(1000, L, 1000, blocksize).
```

The resulting predictions (Figure IV.2) capture very well the behavior for the most efficient block-sizes (between 48 and 128). For instance, Variant 3 (●) —the fastest— attains its top performance with a block-size of 64 according to both the measurements and our prediction.

The quality of our prediction decreases for very large and very small block-sizes. For our goal —determining the fastest algorithm configuration— the low accuracy in regions with low performance is not an issue. The decreasing accuracy in the maximum and average performance for variant 4 (●), resulting from measurement outliers in the model generation, shows that these quantities are less well suited for predictions. The median on the other hand is very reliable.

3) Shared memory parallelism: Our modeling strategy applies well to shared memory architectures too. In this next experiment, we utilize all the 8 cores of the processor. We achieve parallelism by linking the routines for matrix inversions with the multithreaded version of the MKL library. The performance predictions are obtained from performance models generated from measurements of the multithreaded BLAS routines.

The resulting predictions along with measurements of `trinv`'s performance are shown in Figure IV.3. Understandably, the predictions show a greater spread between the minimum and maximum expected efficiency. While the medians are less accurate than those for the single core setup, the measurements are still matched well by these ranges. Most importantly, for all problem sizes, our predictions allow us to rank the four algorithmic variants correctly.
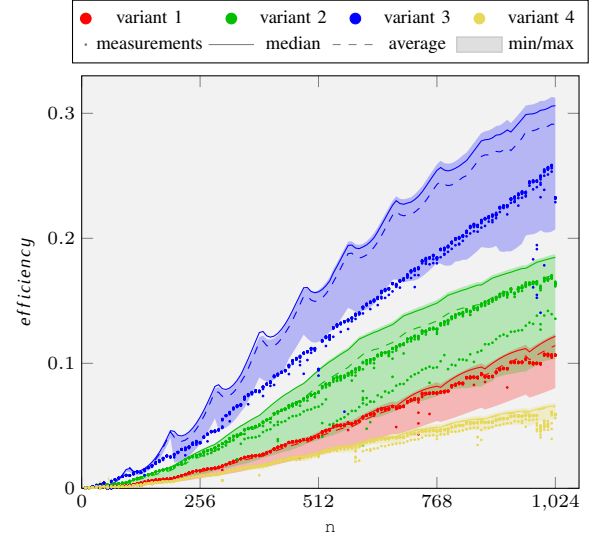
B. Sylvester Equation: Solving $LX + XU = C$ for X

We now study of a more complicated operation: the solution of the Sylvester equation. This operation, encountered in control theory, is generally of the form $AX + XB = C$, where $A \in \mathbb{R}^{m \times m}$, $B \in \mathbb{R}^{n \times n}$, and $C \in \mathbb{R}^{m \times n}$ are given, and $X \in \mathbb{R}^{m \times n}$ is to be computed. We consider a special case, where A and B are lower and upper triangular matrices, respectively: $LX + XU = C$.

With CLICK [5], [6], a tool for the automatic generation of blocked algorithms, we generated code for 16 algorithmic variants. Each of them takes as input the three matrices L, U, and X; X initially contains the input matrix C and is overwritten with the solution to the equation.

The exemplary update statements for variants 1 and 16 are given below. There, $\Omega(L, U, X_{ij})$ denotes a recursive invocation to the Sylvester equations solver for the smaller matrix X_{ij}. The signature of this solver is `sylvi(m, n, L, ldL, U, ldU, X, ldX, blocksize)` with $i \in \{1, \dots, 16\}$.

Variant 1	Variant 16
$X_{01} \leftarrow X_{01} - X_{00}U_{01}$	$X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$
$X_{10} \leftarrow X_{10} - L_{10}X_{00}$	$X_{12} \leftarrow X_{12} - X_{11}U_{12}$
$X_{01} \leftarrow \Omega(L_{00}, U_{11}, X_{01})$	$X_{21} \leftarrow X_{21} - L_{21}X_{11}$
$X_{10} \leftarrow \Omega(L_{11}, U_{00}, X_{10})$	$X_{12} \leftarrow \Omega(L_{11}, U_{22}, X_{12})$
$X_{11} \leftarrow X_{11} - X_{10}U_{01}$	$X_{21} \leftarrow \Omega(L_{22}, U_{11}, X_{21})$
$X_{11} \leftarrow X_{11} - L_{10}X_{01}$	$X_{22} \leftarrow X_{22} - X_{21}U_{12}$
$X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$	$X_{22} \leftarrow X_{22} - L_{21}X_{12}$

These blocked algorithms differ from those for `trinvi` in in a number of ways.

- They operate on three matrices, overwriting one of them with the output.
- The input matrices are of different sizes, and not all of them are square: $L \in \mathbb{R}^{m \times m}$, $U \in \mathbb{R}^{n \times n}$, and $X \in \mathbb{R}^{m \times n}$. Moreover, the matrices are traversed along the diagonal as far as possible and then along the remaining dimension.
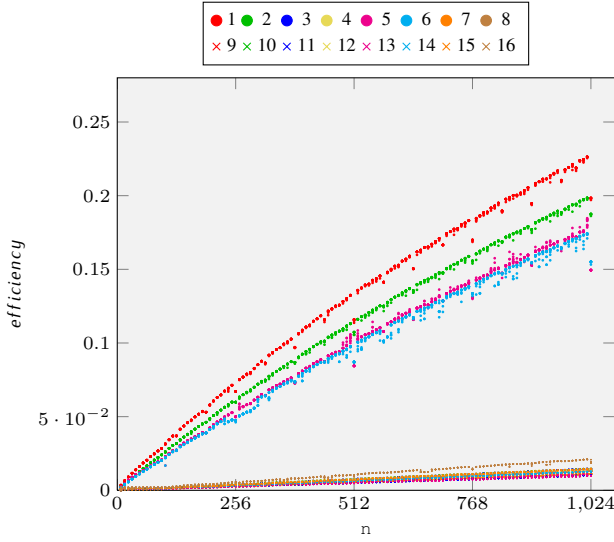- At each iteration, the algorithms perform three recursive
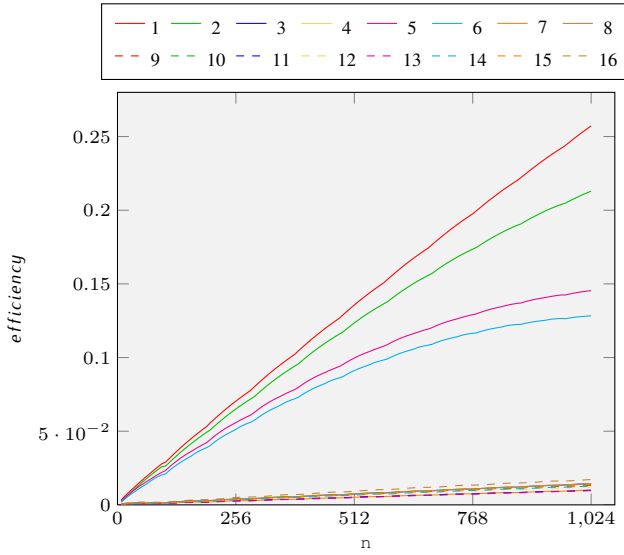
Fig. IV.4: `sylv`: Performance observations.

Fig. IV.5: `sylv`: Performance predictions.

calls to Ω. These operate not only on the $X_{11} \in \mathbb{R}^{\text{blocksize} \times \text{blocksize}}$ but also on the matrix panels X_{01}, X_{10}, X_{12}, and X_{21}. For the latter, our C implementation invokes the blocked algorithms recursively; only the small matrices X_{11} trigger their unblocked versions.

In our tests, we consider the case

```
            m   n   L   ldL   U   ldU   X   ldX   blocksize
sylvi( n,  n,  L,   n,  U,   n,  X,   n,    96   ).
```

All matrices are of size n × n, n ∈ {8, 16, ..., 1024} and we use 96 as block-size. Figures IV.4 and IV.5 compare our predictions for these algorithms with corresponding measurements of their implementations, where

$$efficiency = \frac{n^3 + n^2}{2ticks}.$$

We observe significantly different performances across algorithms: At n = 1024 variant 1 (—) is more than 20 times faster than variant 13 (- -). Indeed, twelve of the variants

attain a performance below 2%, while the other four reach values around 20%. In such a scenario, it is first crucial to tell apart the two groups, and then to correctly rank the four top variants. Although our individual predictions are not especially accurate, they fulfill the objective perfectly, separating the groups, and ordering variants 1 (—), 2 (—), 5 (—), and 6 (—) as the top most efficient algorithms.

V. CONCLUSION

In this article, we presented an approach to analyze and model the performance of dense linear algebra routines. Our goal was to rank a given collection of blocked algorithms according to their performance and to optimize their configuration, without executing them. Towards this goal, we created a performance modeling tool, the Modeler, that automatic generates models for BLAS and LAPACK routines. We introduced two strategies to originate piecewise polynomial models, to favor either speed or accuracy. Upon creation, the set of models is stored in an easily accessible repository, for easy access and evaluation. In order to predict the performance of a blocked algorithm, the performance models of its building blocks are then evaluated and combined.

We showed that the approach is applicable to operations with numerous algorithm variants both on single- and multi-core systems; experiments confirmed that our predictions are able to both correctly tell apart the variants according to their performance, and to identify the optimal algorithmic block-size.

REFERENCES

[1] J. Cuenca, D. Giménez, and J. González, "Architecture of an automatically tuned linear algebra library," *Parallel Comput.*, vol. 30, no. 2, pp. 187–210, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2003.11.002

[2] J. Dongarra and P. Luszczek, "Reducing the time to tune parallel dense linear algebra routines with partial execution and performance modelling," University of Tennessee Computer Science Technical Report, Tech. Rep., 2010.

[3] R. Iakymchuk and P. Bientinesi, "Modeling performance through memory-stalls," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 2, 2012, to appear.

[4] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[5] D. Fabregat-Traver and P. Bientinesi, "Automatic generation of loop-invariants for matrix operations," in *Computational Science and its Applications, International Conference*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 82–92.

[6] ——, "Knowledge-based automatic generation of partitioned matrix expressions," in *Computer Algebra in Scientific Computing*, ser. Lecture Notes in Computer Science, V. Gerdt, W. Koepf, E. Mayr, and E. Vorozhtsov, Eds., vol. 6885. Springer Berlin / Heidelberg, 2011, pp. 144–157.